SpaceWire RMAP Library v2

User Guide

Takayuki Yuasa

Japan Aerospace Exploration Agency, Institute of Space and Astronautical Science

yuasa _at_mark_ astro.isas.jaxa.jp

January 10, 2012

Contents

1 Overview of SpaceWire RMAP Library

SpaceWire RMAP Library is an open-source C++ class library for developments and tests of SpaceWire[1] networks and data transfer over RMAP[2]. The library is highly modularized, and provides easy-to-use access to SpaceWire interfaces and a software RMAP stack.

[1] ECSS-E-ST-50-12C.

[2] Remote Memory Access Protocol. ECSS-E-ST-50-52C.

The SpaceWireIF class abstracts a physical SpaceWire interface, and an implementation class for SpaceWire-over-TCP interface is implemented as SpaceWireIFOverTCP. RMAP initiator and target functions can be multiplexed on a single SpaceWire interface, and the RMAPEngine class acts as a central engine for RMAP-related activities. RMAP target node information such as a target logical address, a target SpaceWire address, key, and so on, are managed via XML-like configuration files making it easier to handle multiple target nodes in a large SpaceWire network.

SpaceWire RMAP Library can be used on Mac OS X and Linux (probably with slight modification), and perhaps Windows with the Cygwin environment. If anyone ports the library to Windows, feedback the resulting source tree. SpaceWire RMAP Library uses libraries listed below:

- CxxUtilities[3]
- XMLUtilities by Soki Sakurai from The University of Tokyo[4]
- xerces-c++ by the Apache project[5]

Among these, CxxUtilities and XMLUtilities are header-only libraries, and and xerces-c++ should be built before using SpaceWireRMAPLibrary.

Note that the software is distributed for free assuming that this is useful for some users without any warranty or official support, and developers are not responsible for any damages caused by this software.

1.1 Changes from SpaceWire RMAP Library v1

In 2006, SpaceWire RMAP Library v1 was released, and has been used in many applications and in many institutes. Since RMAP was in a drafting phase at that time, the RMAP implementation in v1 was tentative, and is now obsolete in many ways (e.g. naming convention). SpaceWire RMAP Library v2 which was written from scratch totally replaces v1, with many new functions which improves flexibility and controllability of SpaceWire and RMAP functions.

The SpaceWire RMAP Library v1 code is still contained in the v2 folder so as to allow old users to work with their applications developed for v1 (see SpaceWireRMAPLibrary/classic/). However, maintenance to the v1 code is suspended, and development power is devoted to v2.

2 About this user guide

The user guide is provided *as is* expecting that this is somewhat useful for users to use the software. This is a voluntary mission, and therefore, kind help is always welcome. It is greatly appreciated to make contributions by feed-backing comments, revising documents, and so on.

One of the Japanese authors of this document, Takayuki Yuasa, is sorry for his limited English capability, and will be very happy if anyone can help to improve it. Comments on grammar, vocabularies, phrasing, and composition are welcome!

2.1 Latest information and feedback

Latest information on SpaceWire RMAP Library can be found at the open-source SpaceWire project website [6].

2.2 References

Documents listed below can be a nice reference when better understanding SpaceWire RMAP Library. Some of the documents can be obtained from the open-source SpaceWire project website[6].

- SpaceWire-to-GigabitEther User Guide
- SpaceWire RMAP GUI User Guide
- ECSS-E-ST-50-12C - ”SpaceWire - Links, nodes, routers and networks” by ECSS
- ECSS-E-ST-50-52C ”SpaceWire - Remote memory access protocol” by ECSS

2.3 Revisions

- 2012-01-10 Takayuki Yuasa. First release.

[3] `https://github.com/yuasatakayuki/CxxUtilities`
[4] `https://github.com/sakuraisoki/XMLUtilities/`
[5] `http://xerces.apache.org/xerces-c/`
[6] The open-source SpaceWire project: `https://galaxy.astro.isas.jaxa.jp/~yuasa/SpaceWire`

3 Download, install, and compile with user applications

SpaceWire RMAP Library is distributed as a zip archive at the open-source SpaceWire project website. Git repository can be cloned from the github page[7]. Since SpaceWire RMAP Library is a header-only library, installation is simple; unzip the archive, and then move SpaceWireRMAPLibrary folder to wherever you like. Possible locations are for example /Users/*username*/Documents/workspace/SpaceWireRMAPLibrary, /Users/*username*/install/SpaceWireRMAPLibrary, or /usr/local/SpaceWireRMAPLibrary.

After installing the library, an environmental variable ⊠gSPACEWIRERMAPLIBRARY_PATH⊠h should be set to point the installed folder to simplify Makefiles used when compiling user applications. In the shell initialization file (.zshrc for zsh, and .bashrc for bash), add a line below:

Listing 1: Sample code for filling the size and the data sections.

```
export SPACEWIRERMAPLIBRARY_PATH=/Users/yuasa/workspace/SpaceWireRMAPLibrary
```

Change the path to match your install location.

The library uses several other libraries as mentioned in §1. Install xerces-c++ by downloading its source archive from the project page[5]. An example Makefile distributed with SpaceWire RMAP Library uses an environmental variable "XERCESDIR". Set "XERCESDIR" in the shell initialization file, e.g.

Listing 2: Sample code for filling the size and the data sections.

```
export XERCESDIR=/Users/yuasa/work/install/xerces-c-3.1.1
```

Although CxxUtilities and XMLUtilities are also required by SpaceWire RMAP Library, a release version of SpaceWire RMAP Library includes these libraries for users' convenience; in SpaceWireRMAPLibrary/externalLibraries/. Therefore, by default, environmental variables for these libraries are automatically set in an example Makefile, and users do not need to redefine them. If a user wants to use their own installation(s) of CxxUtilities and/or XMLUtilities, set "CXXUTILITIES_PATH" and "XMLUTILITIES_PATH" reflecting his/her environment.

Table 1 lists environmental variables used by SpaceWire RMAP Library Makefile. List 3 presents an example Makefile which can be used to compile a user application with SpaceWire RMAP Library and related libraries.

Table 1: List of environmental variables used in SpaceWire RMAP Library Makefile.

Name	Value	Example
SPACEWIRERMAPLIBRARY_PATH	path to SpaceWire RMAP Library	/Users/yuasa/workspace/SpaceWireRMAPLibrary
XERCESDIR	path to xerces-c++ installation	/Users/yuasa/workspace/xerces-c-3.1.1
CXXUTILITIES_PATH (optional)	path to CxxUtilities	/Users/yuasa/workspace/CxxUtilities
XMLUTILITIES_PATH (optional)	path to XMLUtilities	/Users/yuasa/workspace/XMLUtilities

Listing 3: Sample Makefile.

```
################################################################
#An example Makefile for SpaceWire RMAP Library.
################################################################

#Note 1:
#To compile a user application with SpaceWire RMAP Library,
#set SPACEWIRERMAPLIBRARY_PATH and XERCESDIR in the shell
#initialization file first.
#
#Execute below to check if these variables are correctly
#set in your shell.
#
# > ls $SPACEWIRERMAPLIBRARY_PATH
# > ls $XERCESDIR
#
#If no error is observed, the paths seem valid.

#Note 2:
#This Makefile assumes a user-application source code named
#UserApplication.cc. If other source files, include paths,
#and/or linker flags are necessary for compile, add them to
#CXXFLAGS and LDFLAGS.
```

[7] https://github.com/yuasatakayuki/SpaceWireRMAPLibrary

```
#################################################################


#Set target (binary names)
#See also the rule part below.
TARGETS = \
UserApplication

#Check CxxUtilities
ifndef $(CXXUTILITIES_PATH)
CXXUTILITIES_PATH = $(SPACEWIRERMAPLIBRARY_PATH)/externalLibraries/CxxUtilities
endif

#Check XMLUtilities
ifndef $(XMLUTILITIES_PATH)
XMLUTILITIES_PATH = $(SPACEWIRERMAPLIBRARY_PATH)/externalLibraries/XMLUtilities
endif

#Set compiler/linker flags
CXXFLAGS = -I$(SPACEWIRERMAPLIBRARY_PATH)/includes -I$(CXXUTILITIES_PATH)/includes -I$(
    XMLUTILITIES_PATH) -I/$(XERCESDIR)/include
LDFLAGS = -L/$(XERCESDIR)/lib -lxerces-c


TARGETS_OBJECTS = $(addsuffix .o, $(basename $(TARGETS)))
TARGETS_SOURCES = $(addsuffix .cc, $(basename $(TARGETS)))

#################################################################

.PHONY : all

all : $(TARGETS)

UserApplication : UserApplication.o
        $(CXX) -g $(CXXFLAGS) -o $@ $@.cc $(LDFLAGS)

clean :
        rm -rf $(TARGETS) $(addsuffix .o, $(TARGETS))
```

4 Overall structure of SpaceWire RMAP Library

Figure 1 presents a structure diagram of SpaceWire RMAP Library. The SpaceWireIF abstracts real SpaceWire interfaces providing standardized ways of sending/receiving packets and emitting time codes. Upon the SpaceWire layer, the software RMAP stack is implemented. The both layers are contained in a flat source tree in the includes folder of SpaceWireRMAPLibrary/.

4.1 Folders

SpaceWire RMAP Library consists of several folders as described below.

includes contains header files of SpaceWire RMAP Library. In a user-application Makefile, add a path to this folder in the compiler flag.

externalLibraries contains CxxUtilities and XMLUtilities source trees for those who do not have their own installation of these libraries. The attached example Makefile uses these bundled libraries by default.

exampleMakefile contains an example Makefile for a user application which uses SpaceWire RMAP Library. Necessary compiler and linker flags are also described in the file.

sources contains free-gift programs built with SpaceWire RMAP Library, tutorial source code, and test codes.

classic contains obsolete (not maintained) SpaceWire RMAP Library v1 source tree.

4.2 Files

SpaceWire.hh and RMAP.hh are the top-level header files for SpaceWire and RMAP functionalities. Load (`#include`) them in a user application to use SpaceWire RMAP Library. Tutorial given in §5 describes codes written in tutorial_XXX.cc in the sources folder. For details of main_XXX files in the sources folder, see the following section.

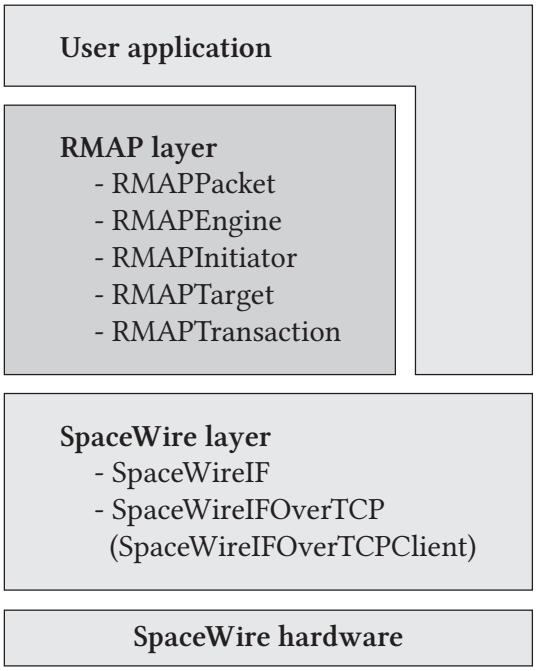

Figure 1: An overall structure of SpaceWireRMAPLibrary.

test_XXX files contained in the sources folder are test codes written for checking implemented library functions. They are left as they are for Interested users' inspection.

4.3 Free-gift programs

In the sources folder, there are several main_XXX.cc files. These programs do very simple tasks using SpaceWire RMAP Library. In ground experiments using SpaceWire and RMAP, these types of simple, standalone tasks are greatly powerful, and therefore, the developers bundled them as free gifts for users.

main_RMAP_calculateCRC calculates CRC for an input byte sequence.

main_RMAP_instructionToString interprets single-byte RMAP instruction field value, and dumps its meaning.

main_RMAP_interpretAsAnRMAPPacket tries to interpret a provided byte sequence as an RMAP packet. When the interpretation is successful, resulting properties of the packet will be displayed as text.

main_RMAP_readWriteRMAPTargetNode performs simple RMAP read/write access to a specified RMAP target node.

main_RMAP_replyStatusToString converts a reply status value to string.

5 Tutorial

In the following subsections, a short walk-through of SpaceWire RMAP Library is described assuming that RMAP initiator is the most interested function for many users. Detailed usages and application specific topics related to SpaceWire and RMAP functions are presented in §6.

For SpaceWire- and RMAP-layer tutorials, refer to tutorial source codes sources/tutorial_SpaceWire.cc and sources/-tutorial_RMAP.cc. To compile these files, set the environmental variables (§3), and then execute make in the SpaceWireRMAPLibrary/source/ folder.

5.1 Use SpaceWire RMAP Library

Include "SpaceWire.hh" and "RMAP.hh" for loading all necessary header files related to the SpaceWire and the RMAP layers. Individual header files could be separately included for applications with limited usage of SpaceWire RMAP

Library; e.g. a user application which only uses the RMAPPacket class, to include "RMAPPacket.hh" may be sufficient. Note that, practically, inclusion of "RMAP.hh" automatically includes "SpaceWire.hh".

Classes defined in SpaceWire RMAP Library are not enclosed with any namespace (i.e. declared at the root level). However, classes of CxxUtilities are declared inside the namespace "CxxUtilities", and therefore, to use them, specify the full path of the class e.g. "CxxUtilities::Condition" or do "using namespace CxxUtilities;" in your source file. Since Thread is a member of CxxUtilities, users may need to put "CxxUtilities::" when constructing a subclass of Thread (note "public CxxUtilities::Thread" not "public Thread").

```
class SubclassOfThread : public CxxUtilities::Thread {
public:
        void run(){
                ... thread content ...
        }
};
```

5.2 Summary of a virtual SpaceWire interface class

List 4 summarizes frequently used user-side interface provided by the SpaceWireIF class. See "SpaceWireIF.hh" for full details of each method. Implementation of virtual methods are given in SpaceWireIFXXX.cc, such as SpaceWireIFOverTCP-Client.hh.

Listing 4: Summary of methods defined in SpaceWireIF.

```
class SpaceWireIF {
public:
/* open/close */
  virtual void open() throw (SpaceWireIFException);
  virtual void close() throw (SpaceWireIFException);

/* send methods */
  virtual void
  send(uint8_t* data, size_t length, SpaceWireEOPMarker::EPPType eopType = SpaceWireEOPMarker::EOP) throw (
       SpaceWireIFException);
  virtual void
  send(std::vector<uint8_t>& data, SpaceWireEOPMarker::EPPType eopType = SpaceWireEOPMarker::EOP) throw (
       SpaceWireIFException);
  virtual void
  send(std::vector<uint8_t>* data, SpaceWireEOPMarker::EPPType eopType = SpaceWireEOPMarker::EOP) throw (
       SpaceWireIFException);

/* receive methods */
  //fast
  virtual std::vector<uint8_t>* receive() throw (SpaceWireIFException);
  //fast
  virtual void receive(std::vector<uint8_t>* buffer) throw (SpaceWireIFException);
  //slow; not recommended
  virtual void
  receive(uint8_t* buffer, SpaceWireEOPMarker::EPPType& eopType, size_t maxLength, size_t& length)  throw (
       SpaceWireIFException);

/* set receive timeout */
  virtual void setTimeoutDuration(double microsecond) throw (SpaceWireIFException);

/* emit timecode */
  virtual void emitTimecode(uint8_t timeIn, uint8_t controlFlagIn = 0x00) throw (SpaceWireIFException);

/* Action related to timecode */
  void addTimecodeAction(SpaceWireIFActionTimecodeScynchronizedAction* action);
  void registerTimecodeAction(SpaceWireIFActionTimecodeScynchronizedAction* action);
  void deleteTimecodeAction(SpaceWireIFActionTimecodeScynchronizedAction* action);
  void clearTimecodeSynchronizedActions();

/* Action related to link close event */
  void addSpaceWireIFCloseAction(SpaceWireIFActionCloseAction* spacewireIFCloseAction);
  void deleteSpaceWireIFCloseAction(SpaceWireIFActionCloseAction* spacewireIFCloseAction);
  void invokeSpaceWireIFCloseActions();

/* EOP/EEP related */
  bool isTerminatedWithEEP();
  bool isTerminatedWithEOP();
  void setReceivedPacketEOPMarkerType(int eopType);
  int getReceivedPacketEOPMarkerType();
  void eepShouldBeReportedAsAnException();
  void eepShouldNotBeReportedAsAnException();
};
```

5.3 Opening/closing a SpaceWire interface

SpaceWire RMAP Library provides a virtual interface for physical SpaceWire devices as defined in the super class SpaceWireIF.hh. Classes named SpaceWireIFXXXX implements interface for real devices such as SpaceWire-to-GigabitEther (i.e. SpaceWireIFOverTCPClient).

The super class defines a method name open() which opens a real SpaceWire interface device, and should be invoked when starting to use the device. For example, in the case of SpaceWire-to-GigabitEther, use sentences below to construct an instance, and open the device.

Practically, SpaceWireIFOverTCPClient throws an exception when timeout occurs. The example below tries to open the device (using the specified IP address), and the open() sentence is enclosed with a try-catch block to detect failure of opening a connection. The default port number is 10030, but this may not be always appropriate for different SpaceWire-to-GigabitEther. See user manual of your device. (for example, Shimafuji's SpaceWire-to-GigabitEther can accept 10031 as well for an additional SpaceWire-to-TCP/IP port)

Listing 5: Sample code for opening SpaceWire-to-GigabitEther.

```
/* Open the SpaceWire interface */
cout << "Opening SpaceWireIF...";
SpaceWireIF* spwif = new SpaceWireIFOverIPClient("192.168.1.100", 10030);
try {
    spwif->open();
} catch (...) {
    cerr << "Connection timed out." << endl;
    exit(-1);
}
cout << "done" << endl;

... user process using spwif ...

/* Close */
spwif->close();
```

5.4 Sending/receiving SpaceWire packet

Three types of send methods are available. The only difference is a type of data container; C-array or std::vector. Basic data type of SpaceWire RMAP Library is uint8_t, and therefore containers should be uint8_t* or std::vector<uint8_t>. Vectors can be passed as a reference or a pointer (the two ways result almost the same speed).

Parameters of the send methods are data (data content and length), and the end-of-packet (EOP) marker. EOP markers is either of SpaceWireEOPMarker::EOP or SpaceWireEOPMarker::EEP.

When an exception occurs while sending a packet, the send method throws it to allow a user application to handle the situation. The example below just dumps a reason of a thrown exception. Practically, users should think about re-trying to send the packet or to notify the exception to higher layers.

Listing 6: Sample code for sending packets.

```
/* Send packet */
try {
    cout << "Send packet1" << endl;
    uint8_t packet1[] = { 0x0a, 0x0b, 0x0c, 0x0d };
    size_t length1 = 4;
    spwif->send(packet1, length1, SpaceWireIF::EOP);
    cout << "Send packet2" << endl;
    std::vector<uint8_t> packet2;
    packet2.push_back(0xe);
    packet2.push_back(0xf);
    packet2.push_back(1);
    packet2.push_back(2);
    packet2.push_back(3);
    spwif->send(packet2, SpaceWireIF::EOP);
} catch (SpaceWireIFException e) {
    cerr << "Exception when sending a packet." << endl;
    cerr << e.toString() << endl;
    exit(-1);
}
cout << "Send packet done" << endl;
```

List 7 sets a timeout duration for receive wait. Note that implementation of timeout counter depends on SpaceWire interfaces, and precision may not be an order of microsecond.

Listing 7: Sample code for setting a receive timeout duration.

```
/* Set receive timeout */
spwif->setTimeoutDuration(1e6);//1sec timeout duration
```

List 8 shows how to receive packets. In the case of receive, std::vector<uint8_t> is a default data container type since basically the size of a packet is unconstrained in SpaceWire (std::vector supports variable length data content, but C-array does not). Two receive methods which interfaces with std::vector are available as used below. In the first example, a pointer to a newly constructed std::vector instance is returned when a packet is received. After processing the packet content, a user application should delete the instance (see `delete packet3;`) although there is no explicit `new` for this instance in this example (SpaceWireIF class internally constructs the instance). The second example is rather straightforward; an instance of std::vector<uint8_t> is passed to the receive method.

Listing 8: Sample code for receiving packets.

```
/* Receive packet */
cout << "Receive packet3" << endl;
try {
    std::vector<uint8_t>* packet3 = spwif->receive();
    cout << "Receive packet3 done (" << packet3->size() << "bytes)" << endl;
    //delete packet3 instance (it was newly constructed by SpaceWireIF internally,
    //and user should delete it to avoid memory leak.
    delete packet3;
} catch (SpaceWireIFException e) {
    if (e.getStatus() == SpaceWireIFException::Timeout) {
        cerr << "Receive timeout" << endl;
    } else {
        cerr << "Exception when receiving a packet." << endl;
        cerr << e.toString() << endl;
        exit(-1);
    }
}
cout << "Receive packet4" << endl;
try {
    std::vector<uint8_t>* packet4 = new std::vector<uint8_t>();
    spwif->receive(packet4);
    cout << "Receive packet4 done (" << packet4->size() << "bytes)" << endl;
    delete packet4;
} catch (SpaceWireIFException e) {
    if (e.getStatus() == SpaceWireIFException::Timeout) {
        cerr << "Receive timeout" << endl;
    } else {
        cerr << "Exception when receiving a packet." << endl;
        cerr << e.toString() << endl;
        exit(-1);
    }
}
```

5.5 Emitting time codes

In addition to the send/receive packet functions, SpaceWireIF is also able to emit time codes using the emitTimecode(uint8_t timeIn, uint8_t controlFlagIn = 0x00) method as shown in List 9. The timeIn parameter should contain a time code value from 0 to 63. The control flags are configurable to support possible future extensions of SpaceWire.

The example waits for 15.625 ms after sending one time code. Time-code value is incremented up to 63. The for loop consumes approximately 1 second to complete. Note that this is just an example, and time-code frequency is one of the most important parameter in a SpaceWire network. The frequency strongly depends on applications, and check if SpaceWire-to-GigabitEther achieves a required precision of emission frequency, and enough small jitter for your application. See SpaceWire-to-GigabitEther User Guide for details of jitters of time-code emission realized by SpaceWire-to-GigabitEther and SpaceWireIFOverTCPClient.

For periodic timecode emission, a thread class which has a similar code as List 9 in the run() method should be implemented, and started (i.e. call start()). See excerpts in List 10, and tutorial_SpaceWireLayer_periodicTimecodeEmission.cc for full details.

Listing 9: Sample code for emitting time codes.

```
    /* Emit timecode */
    cout << "Emit timecode 64times" << endl;
    Condition c;
    try {
        for (uint8_t timecodeValue = 0; timecodeValue < 64; timecodeValue++) {
            cout << "Emitting timecode " << (uint32_t) timecodeValue << endl;
            spwif->emitTimecode(timecodeValue);
            c.wait(1.0 / 64.0); //wait 15.625ms
        }
    } catch (SpaceWireIFException e) {
        cerr << "Exception when receiving a packet." << endl;
        cerr << e.toString() << endl;
        exit(-1);
    }
```

Listing 10: Sample code for periodically emitting time codes.

```
class TimecodeThread: public CxxUtilities::StoppableThread {
private:
    SpaceWireIF* spwif;

public:
    const static double TimecodeFrequency = 64; //Hz

public:
    TimecodeThread(SpaceWireIF* spwif) {
        this->spwif = spwif;
    }

public:
    void run() {
        uint8_t timecode = 0x00;
        while (!isStopped()) {
            try {
                spwif->emitTimecode(timecode);
            } catch (...) {
                using namespace std;
                cerr << "Timecode emission failed" << endl;
            }
            if (timecode == 63) {
                timecode = 0;
            } else {
                timecode++;
            }
            sleep(1 / TimecodeFrequency);
        }
    }
};
```

5.6 Summary of RMAP-related classes

For initiating RMAP transactions, user applications can use RMAPEngine and RMAPInitiator. Information of an RMAP target node is handled being contained in an RMAPTargetNode instance. For accepting RMAP commands and responding to them, subclasses of the RMAPTarget class can utilized with RMAPEngine. Figure 2 shows a overall design of the SpaceWire and RMAP protocol stack used in SpaceWire RMAP Library.

RMAPEngine works as a central engine of the RMAP functionality of a user application. The tasks done by RMAPEngine includes issuing RMAP command packets, managing outstanding RMAP transactions, processing RMAP replies, and responding to incoming RMAP commands. RMAPInitiator bridges RMAPEngine and a user application providing easy-to-use read/write methods which implements RMAP read/write accesses. RMAPTargetNode is used to pass necessary access information, such as target logical address, target SpaceWire address, and key of an accessed RMAP target node, to the read/write methods of RMAPInitiator. Note that RMAPTargetNode corresponds to the RMAPDestination class defined in SpaceWire RMAP Library v1, but with many additional capabilities, particularly interface to an XML-line configuration file (see Appendix B).

Lists 11, 12 and 13 summarizes a part of methods defined in the classes. Less used methods are not shown, and therefore, refer RMAPEngine.hh and RMAPInitiator.hh for full details.

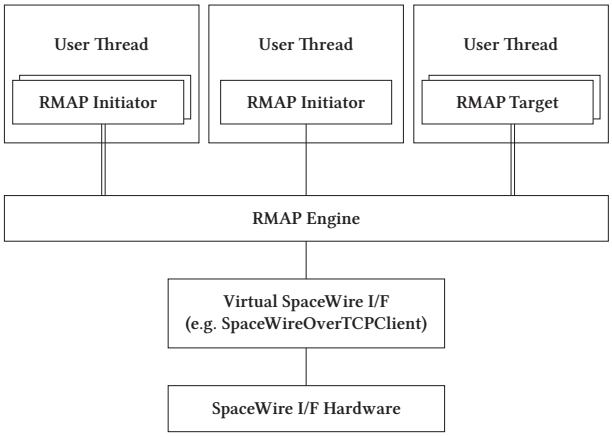

Figure 2: SpaceWire and RMAP protocol stack in SpaceWire RMAP Library.

Listing 11: RMAPEngine methods (excerpts).

```
class RMAPEngine: public CxxUtilities::Thread {
public:

/* constructor */
    RMAPEngine(SpaceWireIF* spwif);

/* start/stop */
    virtual void start();
    void stop();
    bool isStopped();
    bool isStarted();

/* methods used by RMAPInitiator */
    void initiateTransaction(RMAPTransaction* transaction) throw (RMAPEngineException);
    void cancelTransaction(RMAPTransaction* transaction) throw (RMAPEngineException);

/* raw packet send method which even can be used while RMAPEngine is running */
    void sendPacket(std::vector<uint8_t>* bytes);

/* methods used when a user application implements an RMAPTarget */
    void addRMAPTarget(RMAPTarget* rmapTarget);
    void removeRMAPTarget(RMAPTarget* rmapTarget);

/* accessor for a SpaceWireIF instance */
    void setSpaceWireIF(SpaceWireIF* spwif);
    SpaceWireIF* getSpaceWireIF();

/* actions invoked when RMAPEngine is stopped (automatically or manually) */
    void addRMAPEngineStoppedAction(RMAPEngineStoppedAction* rmapEngineStoppedAction);
    void removeRMAPEngineStoppedAction(RMAPEngineStoppedAction* rmapEngineStoppedAction);
    CxxUtilities::Actions* getRMAPEngineStoppedActions();
};
```

Listing 12: RMAPInitiator methods (excerpts).

```
class RMAPInitiator {
public:
/* constructor */
    RMAPInitiator(RMAPEngine *rmapEngine);

/* RMAP Read methods */
    /* fast */
    void read(RMAPTargetNode* rmapTargetNode, uint32_t memoryAddress,
            uint32_t length, uint8_t *buffer, double timeoutDuration = DefaultTimeoutDuration)
            throw (RMAPEngineException, RMAPInitiatorException, RMAPReplyException);

    /* easy to use, but somewhat slow due to data copy. */
    /* this methods returns a pointer to a newly constructed std::vector instance */
    std::vector<uint8_t>* readConstructingNewVecotrBuffer(std::string targetNodeID,
            std::string memoryObjectID, double timeoutDuration = DefaultTimeoutDuration)
            throw (RMAPEngineException, RMAPInitiatorException, RMAPReplyException);
```

```
17
18      /* convenient, but somewhat slow due to RMAPTargetNode DB and  RMAPMemoryObject DB search */
19      void read(std::string targetNodeID, std::string memoryObjectID, uint8_t* buffer,
20              double timeoutDuration = DefaultTimeoutDuration)
21              throw (RMAPEngineException, RMAPInitiatorException, RMAPReplyException);
22
23      /* convenient, but somewhat slow due to RMAPTargetNode DB search */
24      void read(std::string targetNodeID, uint32_t memoryAddress, uint32_t length,
25              uint8_t* buffer, double timeoutDuration = DefaultTimeoutDuration)
26              throw (RMAPEngineException, RMAPInitiatorException, RMAPReplyException);
27
28      /* convenient, but somewhat slow due to RMAPMemoryObject DB search */
29      void read(RMAPTargetNode* rmapTargetNode, std::string memoryObjectID,
30              uint8_t *buffer, double timeoutDuration = DefaultTimeoutDuration)
31              throw (RMAPEngineException, RMAPInitiatorException, RMAPReplyException);
32
33  /* RMAP Write methods */
34      /* fast */
35      void write(RMAPTargetNode *rmapTargetNode, uint32_t memoryAddress,
36              uint8_t *data, uint32_t length, double timeoutDuration = DefaultTimeoutDuration)
37              throw (RMAPEngineException, RMAPInitiatorException, RMAPReplyException);
38
39      /* convenient, but somewhat slow due to RMAPTargetNode DB and  RMAPMemoryObject DB search */
40      void write(std::string targetNodeID, std::string memoryObjectID, uint8_t* data,
41              double timeoutDuration = DefaultTimeoutDuration)
42              throw (RMAPEngineException, RMAPInitiatorException, RMAPReplyException);
43
44      /* convenient, but somewhat slow due to RMAPTargetNode DB search */
45      void write(std::string targetNodeID, uint32_t memoryAddress, uint8_t *data,
46              uint32_t length, double timeoutDuration = DefaultTimeoutDuration)
47              throw (RMAPEngineException, RMAPInitiatorException, RMAPReplyException);
48
49      /* convenient, but somewhat slow due to RMAPMemoryObject DB search */
50      void write(RMAPTargetNode *rmapTargetNode, std::string memoryObjectID,
51              uint8_t* data, double timeoutDuration = DefaultTimeoutDuration)
52              throw (RMAPEngineException, RMAPInitiatorException, RMAPReplyException);
53
54
55  /* set/get logical address of this RMAPInitiator */
56      void setInitiatorLogicalAddress(uint8_t initiatorLogicalAddress);
57      uint8_t getInitiatorLogicalAddress();
58
59  /* accessor for other RMAP options */
60      void setReplyMode(bool replyMode);
61      void unsetReplyMode();
62      bool isReplyModeSet();
63
64      void setIncrementMode(bool incrementMode);
65      void unsetIncrementMode();
66      bool isIncrementModeSet();
67
68      void setVerifyMode(bool verifyMode);
69      void unsetVerifyMode();
70      bool isVerifyModeSet();
71
72      void setTransactionID(uint16_t transactionID);
73      void unsetTransactionID();
74      uint16_t getTransactionID();
75      bool isTransactionIDSet();
76
77  /* accessor for raw packet pointer */
78      RMAPPacket* getCommandPacketPointer();
79      RMAPPacket* getReplyPacketPointer();
80
81  /* interface for RMAPTargetNodeDB */
82      void setRMAPTargetNodeDB(RMAPTargetNodeDB* targetNodeDB);
83      RMAPTargetNodeDB* getRMAPTargetNodeDB();
84  };
```

Listing 13: RMAPTargetNode methods (excerpts).

```
1  class RMAPTargetNode: public RMAPNode {
2  public:
3  /* constructor */
```

```
    RMAPTargetNode();

/* interfaces to XML-like configuration file */
    static std::vector<RMAPTargetNode*> constructFromXMLFile(std::string filename)
        throw (XMLLoader::XMLLoaderException, RMAPTargetNodeException, RMAPMemoryObjectException);

    static std::vector<RMAPTargetNode*> constructFromXMLFile(XMLNode* topNode)
        throw (XMLLoader::XMLLoaderException, RMAPTargetNodeException, RMAPMemoryObjectException);

    static RMAPTargetNode* constructFromXMLNode(XMLNode* node)
        throw (XMLLoader::XMLLoaderException, RMAPTargetNodeException, RMAPMemoryObjectException);

/* accessor for options */
    uint8_t getDefaultKey();
    void setDefaultKey(uint8_t defaultKey);

    std::vector<uint8_t> getReplyAddress();
    void setReplyAddress(std::vector<uint8_t>& replyAddress);

    uint8_t getTargetLogicalAddress();
    void setTargetLogicalAddress(uint8_t targetLogicalAddress);

    std::vector<uint8_t> getTargetSpaceWireAddress();
    void setTargetSpaceWireAddress(std::vector<uint8_t>& targetSpaceWireAddress);

    void setInitiatorLogicalAddress(uint8_t initiatorLogicalAddress);
    void unsetInitiatorLogicalAddress();
    bool isInitiatorLogicalAddressSet();
    uint8_t getInitiatorLogicalAddress();

/* dealing with memory objects available on an RMAPTargetNode */
    void addMemoryObject(RMAPMemoryObject* memoryObject);
    std::map<std::string, RMAPMemoryObject*>* getMemoryObjects();

/* accessor for registered memory objects */
    RMAPMemoryObject* getMemoryObject(std::string memoryObjectID)
        throw (RMAPTargetNodeException);
    RMAPMemoryObject* findMemoryObject(std::string memoryObjectID)
        throw (RMAPTargetNodeException);

/* converts an instance to string or XML string */
    std::string toString(int nTabs = 0);
    std::string toXMLString(int nTabs = 0);
};
```

5.7 RMAP read/write using RMAPEngine and RMAPInitiator

RMAPInitiator works with RMAPEngine, and therefore, an RMAPEngine instance should be first constructed, and started to work as List 14 presents. RMAPEngine is a subclass of CxxUtilities::Thread, and has start() method to fork a new thread which waits for incoming packets in the background of the main thread (usually, a user application thread). Since RMAPEngine uses SpaceWireIF, its constructor accepts a pointer to a SpaceWireIF instance. An RMAPInitiator instance should be constructed with a pointer to the RMAPEngine instance. An initiator logical address can be set (the example below just sets the default value 0xFE, but any number 0x20-0xFD could be specified).

Note that multiple RMAPInitiator instances can be constructed, and tied to one RMAPEngine. This allows concurrent multiple transaction using a single SpaceWire interface. There is virtually no limit on the number of RMAPInitiator instances registered to one RMAPEngine. When a user application communicates with many RMAP targets, it is basically strongly recommended to create multiple RMAPInitiator instances and perform read/write transactions concurrently so as to improve bandwidth usage (i.e. for higher data transfer speed).

Listing 14: Sample code for constructing RMAPEngine/RMAPInitiator.

```
/* Construct and start RMAP Engine */
RMAPEngine* rmapEngine = new RMAPEngine(spwif);
rmapEngine->start();

/* Construct an RMAP Initiator instance */
RMAPInitiator* rmapInitiator = new RMAPInitiator(rmapEngine);
rmapInitiator->setInitiatorLogicalAddress(0xFE);
```

List 15 executes RMAP read/write using a manually constructed RMAPTargetNode instance. Read buffers can be either of C-array and std::vector<uint8_t>. Write data are expected to be passed using C-array (std::vector<uint8_t>::begin() could be used as well). When performing read/write accesses, time-out duration can be passed as a parameter to avoid infinite wait for a reply packet (when target node information is incorrect or an RMAP target is not working, a reply packet may not be received by RMAPEngine, and therefore, generally, RMAPInitiator should terminate wait at a certain point).

RMAP options, such as reply mode, verification mode, address increment mode, and so on, can be set via RMAPInitiator methods (see List 12). Default values of these options can be found (even changed) in RMAPProtocol.hh.

Listing 15: Sample code for performing RMAP read/write using a manually constructed RMAPTargetNode instance.

```
////////////////////////////////////////////////////////////////////////////////
/* Example 1 */
/* Manually sets RMAPTargetNode information */
cout << "Example 1" << endl;

RMAPTargetNode rmapTargetNode1;
rmapTargetNode1.setTargetLogicalAddress(0xfe);
rmapTargetNode1.setDefaultKey(0x20);
std::vector<uint8_t> targetSpaceWireAddress;
targetSpaceWireAddress.push_back(0x01);
targetSpaceWireAddress.push_back(0x0a);
targetSpaceWireAddress.push_back(0x05);
rmapTargetNode1.setTargetSpaceWireAddress(targetSpaceWireAddress);
std::vector<uint8_t> replyAddress;
replyAddress.push_back(0x08);
replyAddress.push_back(0x03);
replyAddress.push_back(0x0f);
rmapTargetNode1.setReplyAddress(replyAddress);
cout << rmapTargetNode1.toString() << endl;
/* RMAP Read/Write with address/length */
try {
    //case 1-1 : using C-array as a read buffer
    uint32_t readLength = 1024;
    uint8_t* readData = new uint8_t[(size_t) readLength];
    uint32_t readAddress = 0xFF801100;
    rmapInitiator->
        read(rmapTargetNode1, readAddress, readLength, readData, readTimeoutDuration);
    //case 1-2 : using std::vector<uint8_t> as a read buffer
    std::vector<uint8_t> readDataVector;
    rmapInitiator->
        read(rmapTargetNode1, readAddress, readLength,
            (uint8_t*)readDataVector.begin(), readTimeoutDuration);

    //case 1-3 : write using C-array write data
    uint32_t writeAddress = 0xFF803800;
    uint32_t writeLength = 4;
    uint8_t* writeData = new uint8_t[writeLength];
    writeData[0] = 0xAB;
    writeData[1] = 0xCD;
    writeData[2] = 0x12;
    writeData[3] = 0x34;
    rmapInitiator->
        write(rmapTargetNode1, writeAddress, writeData, writeLength, writeTimeoutDuration);

    delete readData;
    delete writeData;
    delete rmapTargetNode1;

    cout << "RMAP Read/Write Example1 done" << endl;

} catch (RMAPInitiatorException e) {
    cerr << "RMAPInitiatorException " << e.toString() << endl;
    cerr << "Continue to next example" << endl;
} catch (RMAPReplyException e) {
    cerr << "RMAPReplyException " << e.toString() << endl;
    cerr << "Continue to next example" << endl;
} catch (RMAPEngineException e) {
    cerr << "RMAPEngineException " << e.toString() << endl;
    cerr << "Continue to next example" << endl;
} catch (...) {
    cerr << "Unkown error" << endl;
    exit(-1);
```

```
}
////////////////////////////////////////////////////////////////////////////////
```

5.8 RMAP read/write specifying IDs of RMAP target nodes (using RMAPTargetNodeDB)

RMAPTargetNode instances can be constructed following information described in an XML-like configuration file. RMAPTargetNodeDB is a collection of RMAPTargetNode instances, and the class provides an easy-to-use constructor "RMAPTargetNodeDB:: RMAPTargetNodeDB(std::string filename);" which loads all the RMAPTargetNode defined in the file. List 16 shows how to load a configuration file. Note that RMAPTargetNode information in the XML file can contain information of memory objects on an RMAP target node such as identifier, memory address, length, and access mode (see Appendix B).

An RMAPInitiator instance accepts an instance of RMAPTargetNodeDB as a data base of RMAP target nodes, and read/write methods are invoked with identifiers of an RMAPTargetNode and a memory object on it as used in List 16. Since there should occur database lookups (for RMAPTargetNode and RMAPMemoryObject), these methods are slightly slower than the ones explained in the previous section which uses RMAPTargetNode*, memory address, and length directly. However, these ID-specifying methods are still very useful because of higher reconfigurability, and modularity of the source code; even when a network configuration and register mapping are changed, it is not necessary to modify source codes, but a configuration file can be easily updated to take into account those changes.

When specified RMAPTargetNode ID or memory object ID is not found in RMAPTargetNodeDB, RMAPInitiator will throw RMAPInitiatorException with status of RMAPInitiatorException::NoSuchRMAPTargetNode or RMAPInitiatorException::NoSuchRMAPMemoryObject.

Listing 16: Sample code for performing RMAP read/write using an RMAPTargetNode instance contained in an RMAPTargetNodeDB constructed from an XML-like configuration file.

```
////////////////////////////////////////////////////////////////////////////////
/* Example 2 */
/* Use RMAPTargetNodes constructed from an XML-like configuration file. */
cout << "Example 2" << endl;
if (argc < 2) {
    cerr << "Example2 requires an XML-like configuration file." << endl;
    exit(-1);
}

//check file existence
if (!CxxUtilities::File::exists(argv[1])) {
    cerr << "File " << argv[1] << " does not exist." << endl;
    exit(-1);
}

//construct RMAPTargetNodes from the XML file
std::string filename(argv[1]);
cout << "Constructing RMAPTargetNodes from " << filename << endl;
RMAPTargetNodeDB* rmapTargetNodeDB;
try {
    rmapTargetNodeDB = new RMAPTargetNodeDB(filename);
} catch (RMAPTargetNodeDBException e) {
    cerr << "An exception thrown while loading the XML file " << filename << endl;
    cerr << e.toString() << endl;
    exit(-1);
}

//check the number of entries
if (rmapTargetNodeDB->getSize() == 0) {
    cerr << "No RMAPTargetNode instance was constructed..." << endl;
    exit(-1);
}

//set the db to RMAPInitiator
rmapInitiator->setRMAPTargetNodeDB(rmapTargetNodeDB);

/* RMAP Read/Write with address/length */
try {
    //case 1-1 : read using C-array as a read buffer
    uint32_t readLength = 2;
    uint8_t* readData = new uint8_t[(size_t) readLength];
    rmapInitiator->
```

```
43          read("SpaceWireDigitalIOBoard", "LEDRegister", readData, readTimeoutDuration);
44
45      //case 1-2 : read using std::vector<uint8_t> as a read buffer
46      std::vector<uint8_t> readDataVector(readLength);
47      rmapInitiator->
48          read("SpaceWireDigitalIOBoard", "LEDRegister", &(readDataVector.at(0)),
                readTimeoutDuration);
49
50      //case 1-3 : write using C-array write data
51      uint32_t writeLength = 2;
52      uint8_t* writeData = new uint8_t[writeLength];
53      writeData[0] = 0xFF;
54      writeData[1] = 0xFF;
55      rmapInitiator->
56          write("SpaceWireDigitalIOBoard", "LEDRegister", writeData, writeTimeoutDuration);
57
58      delete readData;
59      delete writeData;
60
61      cout << "RMAP Read/Write Example2 done" << endl;
62
63  } catch (RMAPInitiatorException e) {
64      cerr << "RMAPInitiatorException " << e.toString() << endl;
65      cerr << "Continue to next example" << endl;
66  } catch (RMAPReplyException e) {
67      cerr << "RMAPReplyException " << e.toString() << endl;
68      cerr << "Continue to next example" << endl;
69  } catch (RMAPEngineException e) {
70      cerr << "RMAPEngineException " << e.toString() << endl;
71      cerr << "Continue to next example" << endl;
72  } catch (...) {
73      cerr << "Unkown error" << endl;
74      exit(-1);
75  }
76  ////////////////////////////////////////////////////////////////////////////////
```

5.9 RMAP Packet creation/interpretation

The RMAPPacket class provides integrated functionalities of RMAP packet creation/interpretation. List ?? summarizes representative methods available in RMAPPacket.

Listing 17: Sample code for manually constructing an RMAP packet.

```
1  class RMAPPacket {
2      bool    getDataCRCIsChecked ();
3      bool    getHeaderCRCIsChecked ();
4      void    setDataCRCIsChecked (bool dataCRCIsChecked);
5      void    setHeaderCRCIsChecked (bool headerCRCIsChecked);
6      void    constructHeader ();
7      void    calculateDataCRC ();
8      void    constructPacket ();
9      std::vector< uint8_t >  getPacket ();
10     std::vector< uint8_t > *    getPacketBufferPointer ();
11     void    interpretAsAnRMAPPacket (uint8_t *packet, size_t length) throw (RMAPPacketException);
12     void    interpretAsAnRMAPPacket (std::vector< uint8_t > &data) throw (RMAPPacketException);
13     void    interpretAsAnRMAPPacket (std::vector< uint8_t > *data) throw (RMAPPacketException);
14     void    setRMAPTargetInformation (RMAPTargetNode *rmapTargetNode);
15     void    setRMAPTargetInformation (RMAPTargetNode &rmapTargetNode);
16     bool    isCommand ();
17     void    setCommand ();
18     bool    isReply ();
19     void    setReply ();
20     bool    isWrite ();
21     void    setWrite ();
22     bool    isRead ();
23     void    setRead ();
24     bool    isVerifyFlagSet ();
25     void    setVerifyFlag ();
26     void    unsetVerifyFlag ();
27     void    setVerifyMode ();
28     void    setNoVerifyMode ();
29     bool    isReplyFlagSet ();
30     void    setReplyFlag ();
```

```
    void    unsetReplyFlag ();
    void    setReplyMode ();
    void    setNoReplyMode ();
    bool    isIncrementFlagSet ();
    void    setIncrementFlag ();
    void    unsetIncrementFlag ();
    void    setIncrementMode ();
    void    setNoIncrementMode ();
    uint8_t     getReplyPathAddressLength ();
    void    setReplyPathAddressLength (uint8_t pathAddressLength);
    uint32_t    getAddress ();
    bool    hasData ();
    std::vector< uint8_t >  getData ();
    void    getData (uint8_t *buffer, size_t maxLength) throw (RMAPPacketException);
    void    getData (std::vector< uint8_t > &buffer);
    void    getData (std::vector< uint8_t > *buffer);
    std::vector< uint8_t > *    getDataBuffer ();
    uint8_t     getDataCRC ();
    uint32_t    getDataLength ();
    uint32_t    getLength ();
    uint8_t     getExtendedAddress ();
    uint8_t     getHeaderCRC ();
    uint8_t     getInitiatorLogicalAddress ();
    uint8_t     getInstruction ();
    uint8_t     getKey ();
    uint8_t     getProtocolID ();
    std::vector< uint8_t >  getReplyAddress ();
    uint8_t     getTargetLogicalAddress ();
    std::vector< uint8_t >  getTargetSpaceWireAddress ();
    uint16_t    getTransactionID ();
    void    setAddress (uint32_t address);
    void    setData (std::vector< uint8_t > &data);
    void    setData (uint8_t *data, size_t length);
    void    setDataCRC (uint8_t dataCRC);
    void    setDataLength (uint32_t dataLength);
    void    setLength (uint32_t dataLength);
    void    setExtendedAddress (uint8_t extendedAddress);
    void    setHeaderCRC (uint8_t headerCRC);
    void    setInitiatorLogicalAddress (uint8_t initiatorLogicalAddress);
    void    setInstruction (uint8_t instruction);
    void    setKey (uint8_t key);
    void    setProtocolID (uint8_t protocolID);
    void    setReplyAddress (std::vector< uint8_t > replyAddress, bool
        automaticallySetPathAddressLengthToInstructionField=true);
    void    setTargetLogicalAddress (uint8_t targetLogicalAddress);
    void    setTargetSpaceWireAddress (std::vector< uint8_t > targetSpaceWireAddress);
    void    setTransactionID (uint16_t transactionID);
    uint8_t     getStatus ();
    void    setStatus (uint8_t status);
    uint32_t    getHeaderCRCMode ();
    void    setHeaderCRCMode (uint32_t headerCRCMode);
    uint32_t    getDataCRCMode ();
    void    setDataCRCMode (uint32_t dataCRCMode);
    void    addData (uint8_t oneByte);
    void    clearData ();
    void    addData (std::vector< uint8_t > array);
    std::string     toString ();
    std::string     toXMLString ();
    void    toStringInstructionField (std::stringstream &ss);
    std::string     toXMLStringCommandPacket (int nTabs=0);
    std::string     toXMLStringReplyPacket (int nTabs=0);
};
```

List 18 presents an example of manual packet creation using RMAPPacket excerpted from tutorial _RMAPPacket _creationInterpretation.cc. After setting many options, RMAPPacket::constructPacket() which compiles header, calculates CRCs, and concatenates the header and the data part. Resulting byte sequence can be obtained by calling RMAPPacket:: getPacketBufferPointer() as a std::vector pointer. To display an RMAPPacket, RMAPPacket::toString() or toXMLString() can be utilized. An execution result is shown in List 19.

Listing 18: Sample code for manually constructing an RMAP packet.

```
//Example1 : Manually construct an RMAP packet
vector<uint8_t> targetSpaceWireAddress;
```

```
targetSpaceWireAddress.push_back(3);
targetSpaceWireAddress.push_back(10);
targetSpaceWireAddress.push_back(21);
vector<uint8_t> replyAddress;
replyAddress.push_back(5);
replyAddress.push_back(3);
uint32_t dataLength = 0x31;
RMAPPacket rmapPacket1;
rmapPacket1.setTargetSpaceWireAddress(targetSpaceWireAddress);
rmapPacket1.setReplyAddress(replyAddress);
rmapPacket1.setWrite();
rmapPacket1.setCommand();
rmapPacket1.setIncrementMode();
rmapPacket1.setNoVerifyMode();
rmapPacket1.setExtendedAddress(0x00);
rmapPacket1.setAddress(0xff803800);
rmapPacket1.setDataLength(dataLength);
for (size_t i = 0; i < dataLength; i++) {
    rmapPacket1.addData((uint8_t) i);
}
rmapPacket1.constructPacket();
cout << "RMAPPacket1" << endl;
SpaceWireUtilities::dumpPacket(rmapPacket1.getPacketBufferPointer());
cout << "----------------------------" << endl;
rmapPacket1.setHeaderCRCMode(RMAPPacket::AutoCRC);
rmapPacket1.constructHeader();
cout << rmapPacket1.toString() << endl;
cout << rmapPacket1.toXMLString() << endl;
cout << endl;
```

Listing 19: Result of Example1 of tutorial_RMAPPacket_creationInterpretation.cc.

```
--------- Target SpaceWire Address ---------
0x03 0x0a 0x15
--------- RMAP Header Part ---------
Initiator Logical Address : 0x00
Target Logic. Address     : 0xfe
Protocol ID               : 0x01
Instruction               : 0x65
 ------------------------------
 |Reserved    : 0
 |Packet Type : 1 (Command)
 |Write/Read  : 1 (Write)
 |Verify Mode : 0 (No Verify)
 |Reply Mode  : 0 (No Reply)
 |Increment   : 1 (Increment)
 |R.A.L.      : 1
 |(R.A.L. = Reply Address Length)
 ------------------------------
Key                       : 0x20
Reply Address             : 0x05 0x03
Transaction Identifier    : 0x0000
Extended Address          : 0x00
Address                   : 0xff803800
Data Length (bytes)       : 0x000031 (49dec)
Header CRC                : 0x8b
---------  RMAP Data Part  ---------
[data size = 49bytes]
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f
0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19 0x1a 0x1b 0x1c 0x1d 0x1e 0x1f
0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27 0x28 0x29 0x2a 0x2b 0x2c 0x2d 0x2e 0x2f
0x30
Data CRC                  : 81

Total data (bytes)        : 73
```

List 20 interprets an example byte sequence as an RMAP packet. If RMAPPacket:: interpretAsAnRMAPPacket(uint8_t *packet, size_t length) returns with no exception, the byte sequence is a valid RMAP packet, and interpreted properties are accessible from the RMAPPacket instance. When an exception is thrown, inspect the status and try to disable CRC checks (if RMAPPacketException::InvalidHeaderCRC or RMAPPacketException::InvalidDataCRC is thrown). This can be done via RMAPPacket:: setHeaderCRCIsChecked(bool) or RMAPPacket:: setDataCRCIsChecked(bool). By default, validities of header and data CRCs are checked, and an exception will be thrown when

either or both of them are invalid. List 21 shows an execution result of List 20.

Listing 20: Sample code for manually interpreting an RMAP packet.

```
//Example2 : Interpret a byte sequence as an RMAP packet
RMAPPacket rmapPacket2;
uint8_t bytes[] =
        { 0x07, 0x0B, 0x06, 0x04, 0xFE, 0x01, 0x4F, 0x91,
            00, 00, 00, 00, 00, 00, 00, 0x02, 0x0C, 0x0A,
            0x04, 0x06, 0xFE, 0xAD, 0xDF, 0x00, 0xFF, 0x80, 0x11, 0x00,
            0x00, 0x00, 0x10, 0x2A };
try {
    rmapPacket2.interpretAsAnRMAPPacket(bytes, sizeof(bytes));
} catch (RMAPPacketException e) {
    cerr << "RMAPPacketException " << e.toString() << endl;
    exit(-1);
}
cout << "RMAPPacket2" << endl;
SpaceWireUtilities::dumpPacket(rmapPacket2.getPacketBufferPointer());
cout << "----------------------------" << endl;
rmapPacket2.setHeaderCRCMode(RMAPPacket::AutoCRC);
rmapPacket2.constructHeader();
cout << rmapPacket2.toString() << endl;
cout << rmapPacket2.toXMLString() << endl;
```

Listing 21: Result of Example2 of tutorial_RMAPPacket_creationInterpretation.cc.

```
--------- Target SpaceWire Address ---------
0x07 0x0b 0x06 0x04
--------- RMAP Header Part ---------
Initiator Logical Address : 0xfe
Target Logic. Address     : 0xfe
Protocol ID               : 0x01
Instruction               : 0x4f
 ------------------------------
 |Reserved    : 0
 |Packet Type : 1 (Command)
 |Write/Read  : 0 (Read)
 |Verify Mode : 0 (No Verify)
 |Reply Mode  : 1 (Reply)
 |Increment   : 1 (Increment)
 |R.A.L.      : 3
 |(R.A.L. = Reply Address Length)
 ------------------------------
Key                       : 0x91
Reply Address             : 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x0c 0x0a 0x04 0x06
Transaction Identifier    : 0xaddf
Extended Address          : 0x00
Address                   : 0xff801100
Data Length (bytes)       : 0x000010 (16dec)
Header CRC                : 0x2a
---------  RMAP Data Part  ---------
--- none ---

Total data (bytes)        : 32


<RMAPPacket>
    <ProtocolID>0x01</ProtocolID>
    <InitiatorLogicalAddress>0xfe</InitiatorLogicalAddress>
    <TargetLogicalAddress>0xfe</TargetLogicalAddress>
    <TargetSpaceWireAddress>0x07  0x0b  0x06  0x04</TargetSpaceWireAddress>
    <ReplyAddress>0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x02  0x0c  0x0a  0x04  0x06</
        ReplyAddress>
    <Instruction>0x4f</Instruction>
    <Key>0x91</Key>
    <TransactionIdentifier>0xaddf</TransactionIdentifier>
    <ExtendedAddress>0x00</ExtendedAddress>
    <Address>0xff801100</Address>
    <Length>0x10</Length>
    <HeaderCRC>Auto</HeaderCRC>
    <!-- HeaderCRC = 0x2a (as long as the header is intact) -->
</RMAPPacket>
```

5.10 Multithread and inter-thread communication

6 Detailed usages of the SpaceWire and the RMAP layers

This section, and the following RMAP section, will be updated upon requests from users. If you have any question or comment on a specific function of SpaceWire RMAP Library.

6.1 Handling an interface-close event

A SpaceWire interface might be suddenly closed outside the user control due to several reasons (e.g. disconnection of a TCP/IP socket in SpaceWireIFOverTCPClient). This kind of event should be reported to a user application layer so that it can handle the situation, and not to use the same (closed) interface any more (i.e. not to send/receive packets).

SpaceWireIF uses a call-back framework for notifying such an event to a user application. Relevant class and methods include SpaceWireIFActionCloseAction, and SpaceWireIF:: addSpaceWireIFCloseAction(SpaceWireIFActionCloseAction* spacewireIFCloseAction). Instances of subclasses of SpaceWireIFActionCloseAction can be registered to a SpaceWireIF instance, and these action instances, more precisely its SpaceWireIFActionCloseAction:: doAction (SpaceWireIF*) method, are invoked when SpaceWireIF::close() is called by a user application or by other thread running in the background.

SpaceWire RMAP GUI available from the open-source SpaceWire project uses this call-back mechanism to know SpaceWireIF-close events, and to stop data transfer. See SpaceWire RMAP GUI source code, particularly SpaceWireViewController.h, for practical example. List 22 presents an example of a subclass of SpaceWireIFActionCloseAction defined in SpaceWire RMAP GUI.

Listing 22: An example of a subclass of SpaceWireIFActionCloseAction.

```
class SpaceWireViewContollerCloseActionStopContinuousReceive : public SpaceWireIFActionCloseAction
    {
private:
        id spacewireViewController;
public:
        SpaceWireViewContollerCloseActionStopContinuousReceive(id spacewireViewController){
                this->spacewireViewController=spacewireViewController;
        }

public:
        void doAction(SpaceWireIF* spacewireIF){
                [spacewireViewController stopContinuousPacketReceive];
                [spacewireViewController stopPeriodicTimecodeEmission];
        }
};
```

6.2 Timecode-synchronized action

Timecode-synchronized actions are also implemented as a call-back to registered instances of the SpaceWireIFAction TimecodeScynchronizedAction class (see SpaceWireIF.hh) as was described in the previous section for interface-close event actions.

Users can implement subclasses of SpaceWireIFAction TimecodeScynchronizedAction, and the doAction(unsigned char timecodeValue) method is invoked, via SpaceWireIF::invokeSpaceWireIFCloseActions(), every time when a timecode is received. Filtering of timecode values should be done in the method by users. As presented in List 23, registered timecode-synchronized actions are invoked sequentially, and therefore, time-consuming processes should be avoided in the doAction(unsigned char timecodeValue) method (if time-consuming process should be done, start another thread in the action, and yield main process to the following action instances).

Listing 23: Source code of SpaceWireIF:: invokeSpaceWireIFCloseActions().

```
void invokeSpaceWireIFCloseActions() {
    for (size_t i = 0; i < spacewireIFCloseActions.size(); i++) {
        spacewireIFCloseActions[i]->doAction(this);
    }
}
```

6.3 Change RMAP options

RMAP-related options, such as increment, reply, and verify, can be configured through an RMAPInitiator instance. Considering the increment mode, for example, RMAPInitiator::setIncrementMode(bool) can be used; if the parameter is true, increment bit in the instruction field is set (1), and if false, the increment bit is cleared (0). To restore default setting (defined in RMAPInitiator), RMAPInitiator::unsetIncrementMode() can be invoked. For other options, see RMAPInitiator::setVerifyMode(bool) and RMAPInitiator::setReplyMode(bool).

Transaction ID can be also artificially set via RMAPInitiator::setTransactionID(uint16_t). If this is not set, RMAPEngine automatically puts a certain transaction ID when a command packet is issued, and if this is set, the specified value is used by RMAPEngine. In some cases, the specified transaction ID is already used in RMAPEngine for another transaction, the requested transaction is canceled, and an exception is thrown, RMAPEngineException with a status value of RMAPEngineException::SpecifiedTransactionIDIsAlreadyInUse.

6.4 Handling unexpected events in RMAPEngine

Running RMAPEngine might be stopped by the background process due to, for example, a fatal error in the SpaceWire-interface layer (in reality, RMAPEngine registers a SpaceWireIFActionCloseAction instance to SpaceWireIF to detect inavailability of SpaceWireIF and to stop itself; see the RMAPEngine::RMAPEngineSpaceWireIFActionCloseAction class or List 24). An RMAPEngine-stopped event can be handled using very similar mechanism to the SpaceWireIF-closed event case, using subclasses of RMAPEngineStoppedAction presented in List 25.

Implement a subclass of RMAPEngineStoppedAction, and then register its instance to an RMAPEngine instance via addRMAPEngineStoppedAction(RMAPEngineStoppedAction* rmapEngineStoppedAction). When RMAPEngine::stop() is invoked, registered actions are sequentially called so as to enable a user thread to handle/respond to the stop event. List 26 also shows a practical example of subclass of RMAPEngineStoppedAction, used in SpaceWire RMAP GUI (see RMAPViewController.h contained in the source archive).

This call-back scheme can be expanded to notify another types of events detected in RMAPEngine to a user application, for example, receive of an unexpected RMAP reply packet, or receive of an invalid RMAP packet. If you have any request on addition of call-back, make a feedback to the developers.

Listing 24: Source code of RMAPEngine::RMAPEngineSpaceWireIFActionCloseAction defined in RMAPEngine.hh.

```
1
2 class RMAPEngineSpaceWireIFActionCloseAction: public SpaceWireIFActionCloseAction {
3 private:
4     RMAPEngine* rmapEngine;
5
6 public:
7     RMAPEngineSpaceWireIFActionCloseAction(RMAPEngine* rmapEngine) {
8         this->rmapEngine = rmapEngine;
9     }
10
11 public:
12     void doAction(SpaceWireIF* spwif) {
13         rmapEngine->stop();
14     }
15 };
```

Listing 25: Source code of RMAPEngineStoppedAction defined in RMAPEngine.hh.

```
1 class RMAPEngineStoppedAction: public CxxUtilities::Action {
2 public:
3     virtual void doAction(void* rmapEngine) = 0;
4 };
```

Listing 26: An example of an RMAPEngineStoppedAction subclass used in SpaceWire RMAP GUI.

```
1 class RMAPEngineStoppedActionByRMAPViewController : public RMAPEngineStoppedAction{
2 private:
3         id rmapViewController;
4 public:
5         RMAPEngineStoppedActionByRMAPViewController(id rmapViewController){
6                 this->rmapViewController=rmapViewController;
7         }
8         virtual void doAction(void* rmapEngine){
9                 [rmapViewController rmapEngineWasStopped];
```

```
        }
};
```

A TCP/IP-SpaceWire packet transfer protocol: SSDTP2

SpaceWire has no limitation of the length of the packet, and each SpaceWire packet is terminated using the end of packet character (EOP) or the error end of packet character (EEP). On the other hand, TCP/IP only provides a simple socket which transfers bytes as a stream, and there is no delimiter to handle the end (or dividing point) of the data being transferred. Therefore an encapsulating protocol should be used to transfer SpaceWire packets over a TCP/IP socket. In SpaceWire-to-GigabitEther, a simple header-followed-by-size-and-data protocol is defined and used. The name of the protocol is SSDTP2. In SpaceWire RMAP Library, SpaceWireSSDTPModule supports this protocol.

A.1 Basic structure of SSDTP2 packets

In SSDTP2, encapsulated data have the following structure.
`<Flag 1byte> <Reserved 1byte> <Size 10bytes> <Cargo variable length>`
Flag specifies type of the packet; Data or Control. Data encapsulates SpaceWire packet, and Control contains information needed to control the connection of the SpaceWire-to-TCP/IP converters on both ends of the TCP/IP link (i.e. SpaceWire-to-GigabitEther and a user program on PC). Size contains the size of the Cargo part. The Cargo part can be data or codes which contains control information. Below, the encapsulation structures are described individually.

A.2 Data packet

Table 2 presents the structure of Data packets of SSDTP2. When sending complete SpaceWire packets terminated with EOP (EEP), a Flag value of 0x00 (0x01) is used. Listing 27 shows how to fill the size and the data sections.

When the size of a packet is too long to handle as a single packet, software or hardware logic may divide the packet into multiple segments. In such a case, although the encapsulated packet structure is the same as above, Flag is set at 0x02 to show that the data is segmented and has no end of packet character. Size part should contain the size of the segmented data. After a certain number of un-terminated segments, a terminated segment which has Flag of 0x00 (EEP) or 0x01 (EEP) will complete the whole packet data.

Note that this segmentation has nothing to do with the SpaceWire standard, and arises simply from the difficulty of handling unlimited length of packets in the encapsulating protocol and its implementation. For example when only a small amount of data buffer is available on a SpaceWire-to-TCP/IP converter logic, a SpaceWire packet whose size is larger than the buffer size can not be received without any segmentation. The simple segmentation shown here allows the logic to send a part of the large packet to the TCP/IP side when the buffer becomes full (for example) specifying that the encapsulated data is continued (i.e. not terminated by EOP/EEP). When EOP/EEP is received, the logic can complete sending the segmented data.

Table 2: Structure of a Data packet of SSDTP2.

Flag (see text)	Reserved (0x00)	Size[9]	Size[8]
Size[7]	Size[6]	Size[5]	Size[4]
Size[3]	Size[2]	Size[1]	Size[0]
Data[0]	Data[1]	Data[2]	Data[3]
...	Data[Size-1]		

Listing 27: Sample code for filling the size and the data sections.

```
/* Code to restore the size from the byte array.  */
/* buffer[] should contain the bytes shown above. */
unsigned int size=0;
for(unsigned int i=2;i<12;i++){
    size=size*0x100+buffer[i];
}

/* Code to set the size to the byte array. */
```

```
/* buffer[] should contain the bytes shown above. */
unsigned int size=PacketSize;
for(unsigned int i=11;i>1;i--){
    buffer[i]=size%0x100;
    size=size/0x100;
}
```

A.3 Control packets

Control packets are used for transferring TimeCode and changing setting of SpaceWire-to-GigabitEther.

Encapsulated TimeCode SSDTP2 encapsulates SpaceWire TimeCode so as to allow user programs to emit or receive TimeCode using the SpaceWire-to-GigabitEther device. The encapsulated structure shown in Table 3 is used to encapsulate TimeCode information. Flag is 0x30 when sending TimeCode from a user program to the SpaceWire network via the device, and 0x31 when TimeCode is received at the device from the SpaceWire network. Usually, a user program sends Flag=0x30 and receives Flag=0x31 to/from the device. When Flag=0x31 is received, the user program may perform TimeCode-related operation. Size[0] should be 0x02, and the remaining Size part (Size[1]-Size[9]) should be filled with 0x00. In TimeCode byte, LSB 6bits are used to store 6-bit TimeCode value (time counter value). MSB 2bits are reserved in the standard, and should be "b00".

Changing SpaceWire link speed The Tx link speed of the SpaceWire-to-GigabitEther device can be changed by sending a control packet as presented in Table 4. Flag for this packet is 0x38. The gTxDiv count h specified in the packet is used to divide the original clock of 125 MHz to generate the Tx clock which is fed to SpaceWire IP Transmitter (in the case of the open-source SpaceWire-to-GigabitEther). Users can change the Tx speed more easily by invoking setTxDivCount(unsigned int) method of the SpaceWireIFOverTCPClient class.

Table 3: Timecode encapsulation in SSDTP2.

Flag (see text)	Reserved (0x00)	Size[9] (0x00)	Size[8] (0x00)
Size[7] (0x00)	Size[6] (0x00)	Size[5] (0x00)	Size[4] (0x00)
Size[3] (0x00)	Size[2] (0x00)	Size[1] (0x00)	Size[0] (0x02)
Timecode value	Reserved (0x00)		

Table 4: SSDTP2 Control packet for changing Tx frequency.

Flag (0x38)	Reserved (0x00)	Size[9] (0x00)	Size[8] (0x00)
Size[7] (0x00)	Size[6] (0x00)	Size[5] (0x00)	Size[4] (0x00)
Size[3] (0x00)	Size[2] (0x00)	Size[1] (0x00)	Size[0] (0x02)
TxDiv count	Reserved (0x00)		

B Format of the XML-like configuration file

RMAP Target Node information and memory object information can be stored in an XML-like configuration file. The format is defined in SpaceWire/RMAP Library, specifically, in the RMAPTargetNode class and the RMAPMemoryObject class, and therefore, for details, see SpaceWire/RMAP Library User Guide.

Structures of RMAPTargetNode and RMAPMemoryObject are listed below. When one (or more) of mandatory tag is not found, the configuration file is discarded.

RMAPTargetNode id (name) attribute is mandatory.

TargetLogicalAddress Mandatory.
TargetSpaceWireAddress Mandatory. Array of 0x00-0xFF. (e.g. 0x02 0x0a 0x07 0x01)
ReplyAddress Mandatory. Array of 0x00-0xFF. (e.g. 0x02 0x0a 0x07 0x01)
Key Mandatory. 0x00-0xFF.
InitiatorLogicalAddress Optional. 0x00-0xFF.

RMAPMemoryObject id (name) attribute is mandatory.

ExtendedAddress Optional. Default is 0x00.
Address Mandatory. 0x00000000-0xFFFFFFFF.
Length Mandatory. 0x000000-0xFFFFFF.
Key Optional. 0x00-0xFF. The value defined in the parent RMAPTargetNode is overridden by this value if set.
AccessMode Optional. Any of ReadWrite, ReadOnly, WriteOnly, Readable (=ReadOnly), Writable (=WriteOnly).
IncrementMode Optional. Either of Increment or NoIncrement.

The below shows a template for a configuration file. Note that one file can contain multiple RMAPTargetNodes, and one RMAPTargetNode tag can contains multiple memory object definitions.

Listing 28: Tags which define RMAPTargetNode and RMAPMemoryObject.

```
<root>

<RMAPTargetNode id="NameOfTheRMAPTargetNode">
    <TargetLogicalAddress>0xFE</TargetLogicalAddress>
    <TargetSpaceWireAddress>0x00</TargetSpaceWireAddress>
    <ReplyAddress></ReplyAddress>
    <Key>0x20</Key>
    <InitiatorLogicalAddress>0x35</InitiatorLogicalAddress> <!-- optional -->

    <RMAPMemoryObject id="NameOfTheMemoryObjectOnTheRMAPTargetNode">
        <ExtendedAddress>0x00</ExtendedAddress>
        <Address>0x0000</Address>
        <Length>0x04</Length>
        <Key>0x20</Key> <!-- optional -->
        <IncrementMode>Increment</IncrementMode>
    </RMAPMemoryObject>

    ... other RMAPMemoryObject tags ...

</RMAPTargetNode>

... other RMAPTargetNode tags ...

</root>
```